Remarks/Arguments

The preceding amendments and following remarks are submitted in response to the Office Action of the Examiner mailed December 14, 2005, setting a three month shortened statutory period for response ending March 14, 2005. Claims 1-8 and 10-25 remain pending. Claims 1-3, 10-12 and 25 are allowed, claim 6 is objected to, and claims 4, 5, 7, 8 and 13-24 stand rejected. Reconsideration, examination and allowance of all pending claims are respectfully requested.

In paragraph 4 of the Office Action, the Examiner rejected claims 7 and 8 under 35 U.S.C. § 101 because the claimed invention is directed to non-statutory subject matter. Claim 7 has been amended to recite "[a] computer readable medium encoded with an object for persistent storage". Claims 7-8 are now believed to clearly comply with 35 U.S.C. § 101.

In paragraph 6 of the Office Action, the Examiner rejected claims 7-8 under 35 U.S.C. § 112, second paragraph, as being indefinite for failing to particular point out and distinctly claim the subject matter which applicant regards as the invention. In response, claim 7 has been amended to comply with 35 U.S.C. § 112, second paragraph.

In paragraph 8 of the Office Action, the Examiner rejected claims 4, 5, 7, 8, 13-17 and 19-23 under 35 U.S.C. § 102(e) as being anticipated by U.S. Patent No. 6,125,364 to Greef et al. Turning first to claim 4, the Examiner appears to simply repeat the rejection made in the Final Office Action dated May 31, 2005. In paragraph 15 of the present Office Action, and in response to Applicant's arguments presented in the Preliminary Amendment dated August 31, 2005, the Examiner states:

> The Applicant states that Greef does not disclose providing one or more common interfaces that are used by each of the plurality of objects to write the objects from non-persistent storage to persistent storage. The Applicant further argues that Greef does not teach utilization of IPersistFile, IUunknown, and IPersistStream classes. In response to applicant's argument that the references fail to show the use of these specific COM classes, it is noted that these features upon which applicant relies are not recited in the rejected claim(s). Although the claims are interpreted in light of the specification, limitations from the specification are not read into the claims. See In re Van Geuns, 988 F.2d 1811, USPQ2d 1057 (Fed. Cir. 1993).

(Paragraph 15 of the Office Action dated December 14, 2005). In the Preliminary Amendment dated August 31, 2005, Applicant stated:

> Greef et al. do not appear to disclose or suggest providing one or more common interfaces that are used by each of the plurality of objects to write the objects from non-persistent storage to persistent storage. In fact, the persistent object classes of Greef et al. do not appear to be interfaced based as all.
>
> In one illustrative embodiments of the present invention, the one or more common interfaces may include and be defined for the IPersistFile 20, IUnknown 28, and IPersistStream 36 classes, which are pre-defined by the COM (Component Object Model) specification (see, Specification, page 7, lines 12-22; Figure 1). This illustrative embodiment may be used to, for example, extend component object model (COM) objects to support persistence. This is something that Greef et al. do not teach or contemplate.

(Emphasis Added). As can be seen, Applicant argued that Greef et al. do not appear to disclose or suggest providing one or more common *interfaces* that are used by each of the plurality of objects to write the objects from non-persistent storage to persistent storage. In fact, Applicant argued, the persistent object classes of Greef et al. do not appear to be interfaced based as all. In view of the Examiner comments, it appears the Examiner is not attaching the proper meaning to the term "interface", as used in claim 1. The term "interface" has a particular meaning in the field of object orientated programming. In Java, for example:

> an *interface* is a type, just as a class is a type. Like a class, an interface defines methods. Unlike a class, an interface never implements methods; instead, classes that implement the interface implement the methods defined by the interface. A class can implement multiple interfaces.

(Emphasis Added) (see, page 1, second paragraph, "What Is an Interface?", a copy of which is attached for the Examiner's reference). Likewise, in COM, an interface is a strongly-typed contract between software components to provide a small but useful set of semantically related operations (methods). An interface is the definition of an expected behavior and expected responsibilities. On pages 4-5 of the document entitled "The component Object Model: A Technical Overview" (a copy of which is attached for the Examiner's reference), some interfaces attributes are listed and include:

- **An interface is not a class.** While a class can be instantiated to form a component object, an interface cannot be instantiated by itself because it carries no implementation. A component object must implement that interface and that component object must be instantiated for there to be an interface. Furthermore, different component object classes may implement an interface differently, so long as the behavior conforms to the interface definition (such as two objects that implement **IStack**, where one uses an array and the other a linked list). Thus the basic principle of polymorphism fully applies to component objects.
- **An interface is not a component object.** An interface is just a related group of functions and is the binary standard through which clients and **component object**s communicate. The **component object** can be implemented in any language with any internal state representation, so long as it can provide pointers to interface member functions.

  ...

  It is convenient to adopt a standard pictorial representation for objects and their interfaces. The adopted convention is to draw each interface on an object as a "plug-in jack."

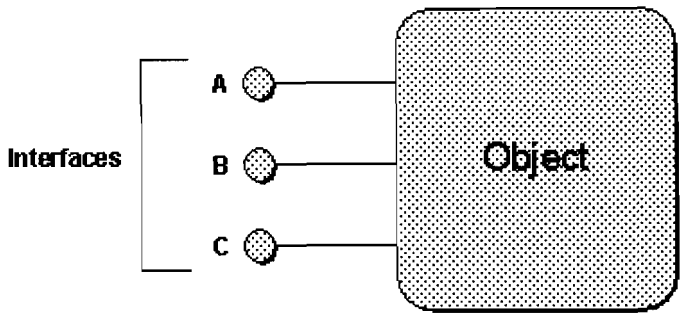

Figure 3. A typical picture of a component object that supports three interfaces A, B, and C.

As can be seen, in the field of object orientated programming, an *interface* is not equivalent to a class or a component object.

The Examiner appears to be equating interfaces with classes (i.e. interface = class). Notable, and in paragraph 15 of the Office Action dated December 14, 2005, the Examiner states:

> As noted in the prior rejection, the Data Base Cursor and the Data Store and commonly utilized classes for streaming objects into and out of persistence, and as such, qualify as common interfaces used by each object according to the broadest reasonable interpretation of the claim language.

However, in the field of object orientated programming, this is simply not correct. One skilled in the object orientated programming art would clearly understand the difference between an *interface* and a *class*. The Examiner acknowledges that the Data Base Cursor and the Data Store are *classes*. Therefore, the Data Base Cursor and the Data Store cannot qualify as common interfaces, as the Examiner suggests. As such, Greef et al. clearly do not suggest the step of "providing one or more common *interfaces* that are used by each of the plurality of objects to write the objects from non-persistent storage to persistent storage", as recited in claim 4. In fact, and as previously argued, the persistent object classes of Greef et al. do not appear to be

interfaced based as all. For these and other reasons, claim 4 is believed to be clearly patentable over Greef et al. For similar reasons, as well as other reasons, claims 5-6, 13-18 and 19-24 are also believed to be clearly patentable over Greef et al.

On page 7 of the Office Action, and with respect to claim 7, the Examiner states that the smart pointer of Greef et al. includes an address attribute for containing the address of an object, and an object unique identifier attribute for containing the unique identifier of an object. With respect to the address attribute, the Examiner states, "as a smart pointer constitutes an object in the system, and all objects in a computer system must contain an address in the computer system, then the smart pointer, while having an identifier for the object it points to, also has an associated address in the computer system for itself" (Emphasis Added).

In response to the § 112 rejection above, claim 7 has been amended to recite:

> 7. A computer readable medium encoded with an object for persistent storage, the object having a smart pointer, wherein the smart pointer includes an address attribute for containing an address of a subject object being pointed to by the smart pointer, *and* an object unique identifier attribute for containing a unique identifier for the subject object being pointed to, wherein the object smart pointer has an assignment operation which correlates the address of the subject object being pointed to and the unique identifier of the subject object being pointed to, and wherein the object includes a load method for using the smart pointer unique identifier attribute to determine and load a new smart pointer address attribute for the subject object being pointed to after the subject object is loaded from persistent storage.

As can be seen, claim 7 now more clearly recites that the smart pointer includes an address attribute for containing an address of a subject object being pointed to by the smart pointer, and an object unique identifier attribute for containing a unique identifier for the subject object being pointed to. As can be seen, the address attribute of claim 7 does not contain an associated

address in the computer system for the smart pointer itself, as the Examiner suggests. Rather, claims 7 recites that the smart pointer includes an address attribute for containing an address of a subject object being pointed to. The Examiner does not allege, nor do Greef et al. appear to disclose, such a smart pointer.

As noted in Applicant's response filed on August 31, 2005, Greef et al. state:

> The Entity Cache comprises collections of "smart pointers" that have been created during a session. FIG. 1 shows the Entity Cache comprising a collection of new entities, a collection of dirty entities and a collection of retrieved entities. Each entity in the system has a unique object identifier and a class identifier as shown in the Entity Class depicted in FIG. 1 [Emphasis Added]. The unique object identifier is generated by the Entity ID Generator shown in FIG. 1. When the object is referenced with the entity identifier, the entity cache swivels the identifier to a smart pointer [Emphasis Added]. If a smart pointer already exists in the entity cache, it returns the smart pointer. If the smart pointer does not exist in the entity cache, the object is loaded from the nonvolatile memory and the smart pointer is returned to the entity in the entity cache. All persistent objects have an entity identifier.
>
> All entity object management is done with smart pointers. "Smart Pointers" are used to handle large amounts of objects in limited memory resources. What is required is a light weight representation of an entity, as described as light weight objects in Gamma et. al. (1995). Each entity has a corresponding smart pointer. It is a sub for a real object in memory. It it has no data members and it overloads the pointer and de-reference operators. All objects manipulate smart pointers. Objects have lists of smart pointers, not pointers to themselves [Emphasis Added].

(Greef et al., column 4, lines 17-41). As can be seen, Greef et al. state that "[a]ll objects manipulate smart pointers. Objects have lists of smart pointers, not pointers to themselves." (Greef et al., column 4, lines 49-41). This suggests that the smart pointers of Greef et al. do not include both an object unique identifier attribute for containing the unique identifier of an object to be pointed to AND an address attribute for containing the address of an object being pointed to, as recited in claim 7. Instead, the smart pointers of Greef et al. appear to only include a

unique identifier, which is generated by the Entity Class depicted in Figure 1 (Greef et al., column 4, lines 23-24). The Entity Cache, Data Store or some other object may generate an actual address for an object being pointed to, but it is not clear from the Greef et al. disclosure.

Greef et al. also state that when the object is referenced with the entity identifier, the entity cache swivels the identifier to a smart pointer. This also indicates that the smart pointers of Greef et al. only include a unique identifier, and that the entity cache is used to swivel the identifier to a smart pointer.

The Examiner specifically points to column 4, lines 52-57 of Greef et al. However, this passage states that when an operation is invoked on a smart pointer, the object that it points to is faulted into the entity cache if it does not already exist. However, this passage also states that this is done by invoking an operation on the Data Store that can find an object in the nonvolatile memory when it is provided with the object's unique identifier. Thus, in this example, the smart pointer appears to store and deliver an object's unique identifier to the Data Store, and it is the Data Store, and not the smart pointer, that finds the object in the non-volatile memory.

In view of the foregoing, Applicant does not believe it can readily be argued that the smart pointers of Greef et al. disclose or suggest both an object unique identifier attribute for containing a unique identifier of an object being pointed to AND an address attribute for containing the address of the object being pointed to, as recited in claim 7.

In addition to the foregoing, Greef et al. do not appear to suggest an object adapted for persistent storage that itself includes a load method for using the smart pointer unique identifier attribute to determine and load a new smart pointer address attribute after the object being

pointed to is loaded from persistent storage, as recited in claim 7. As noted above, it does not appear that the smart pointers of Greef et al. even have an address attribute for containing an address of an object being pointed to. In addition, Greef et al. indicate that it is the Data Store that finds an object in the nonvolatile memory when it is provided with the object's unique identifier, and not a load method of the object itself.

For these and other reasons, claim 7 is believed to be clearly patentable over Greef et al. For similar and other reasons, dependent claim 8 is also believed to be clearly patentable over Greef et al. Claim 9 has been canceled without prejudice.

In addition, and specifically with respect to claim 8, Greef et al. do not appear to suggest an object that is adapted for persistent storage, wherein the object itself includes a stream-out method for streaming out the smart pointer address attribute and unique identifier attribute to persistent storage. In Greef et al., the two primary classes for streaming objects into and out of persistent storage are the Data Base Cursor and the Data Store (see, Greef et al., column 4, lines 42-44). Nothing in the discussion of Figures 7-8 of Greef et al., which relates to storing objects to non-volatile memory, appears to disclose or suggest an object that is adapted for persistent storage, wherein the object itself includes a stream-out method for streaming out the smart pointer address attribute and unique identifier attribute to persistent storage.

In view of the foregoing, Applicant believes that all pending claims 1-8, 10-25 are in condition for allowance. Reexamination and reconsideration are respectfully requested. If the Examiner believes it would be beneficial to discuss the application or its examination in any way, please call the undersigned attorney at (612) 359-9348.

Respectfully submitted,

Dated: March 10, 2006

Brian N. Tufte, Reg. No. 38,638
CROMPTON, SEAGER & TUFTE, LLC
1221 Nicollet Avenue, Suite 800
Minneapolis, MN 55403-2402
Telephone: (612) 677-9050
Facsimile: (612) 359-9349

What Is an Interface?

In general, an *interface* is a device or a system that unrelated entities use to interact. According to this definition, a remote control is an interface between you and a television set, the English language is an interface between two people, and the protocol of behavior enforced in the military is the interface between individuals of different ranks.

Within the Java programming language, an *interface*◆ is a type, just as a class is a type. Like a class, an interface defines methods. Unlike a class, an interface never implements methods; instead, classes that implement the interface implement the methods defined by the interface. A class can implement multiple interfaces.

The bicycle class and its class hierarchy define what a bicycle can and cannot do in terms of its "bicycleness." But bicycles interact with the world on other terms. For example, a bicycle in a store could be managed by an inventory program. An inventory program doesn't care what class of items it manages as long as each item provides certain information, such as price and tracking number. Instead of forcing class relationships on otherwise unrelated items, the inventory program sets up a communication protocol. This protocol comes in the form of a set of method definitions contained within an interface. The inventory interface would define, but not implement, methods that set and get the retail price, assign a tracking number, and so on.

To work in the inventory program, the bicycle class must agree to this protocol by implementing the interface. When a class implements an interface, the class agrees to implement all the methods defined in the interface. Thus, the bicycle class would provide the implementations for the methods that set and get retail price, assign a tracking number, and so on.

You use an interface to define a protocol of behavior that can be implemented by any class anywhere in the class hierarchy. Interfaces are useful for the following:

- Capturing similarities among unrelated classes without artificially forcing a class relationship
- Declaring methods that one or more classes are expected to implement
- Revealing an object's programming interface without revealing its class
- Modeling multiple inheritance, a feature of some object-oriented languages that allows a class to have more than one superclass

The Component Object Model: A Technical Overview

Sara Willliams and Charlie Kindel, Developer Relations Group
Microsoft Corporation

Created: October, 1994

This paper is adapted from an article appearing in Dr. Dobb's Journal, the newsstand special edition, December, 1994.

Note For a complete example of a working component object, please see the COMPPR sample that accompanies this article.

Abstract

This article describes the Component Object Model (COM), a software architecture that allows the components made by different software vendors to be combined into a variety of applications. COM defines a standard for component interoperability, is not dependent on any particular programming language, is available on multiple platforms, and is extensible. This article focuses on the interoperability that COM provides, that is, how components and their clients interact. The sample code provided with this article provides an example of a working component object.

Introduction

The Component Object Model (COM) is a component software architecture that allows applications and systems to be built from components supplied by different software vendors. COM is the underlying architecture that forms the foundation for higher-level software services, like those provided by OLE. OLE services span various aspects of component software, including compound documents, custom controls, inter-application scripting, data transfer, and other software interactions.

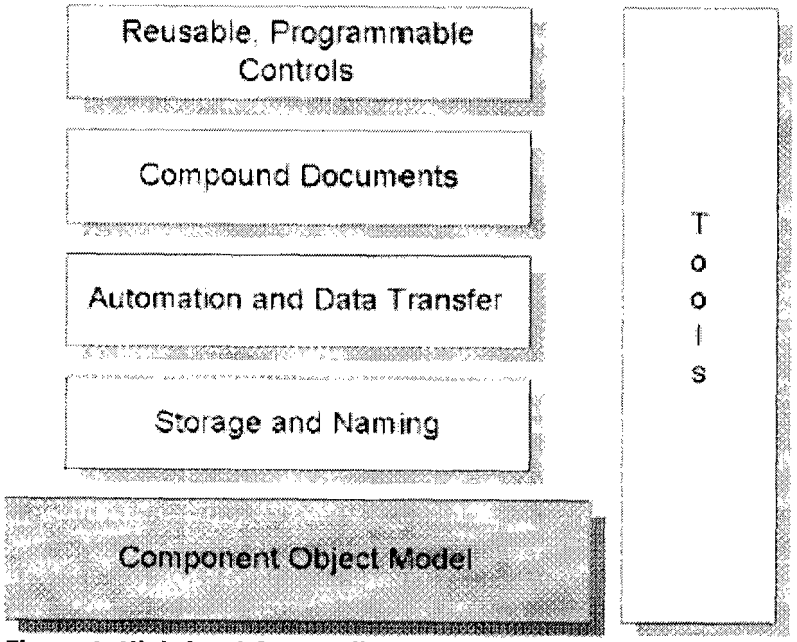

Figure 1. High-level OLE application services are built on the common Component Object Model foundation.

These services provide distinctly different functionality to the user; however, they share a fundamental requirement

for a mechanism that allows binary software components, supplied by different software vendors, to connect to and communicate with each other in a well-defined manner. This mechanism is supplied by COM, a component software architecture that:

- Defines a binary standard for component interoperability
- Is programming language-independent
- Is provided on multiple platforms (Microsoft® Windows®, Microsoft Windows NTÃ‚Â™, Apple® Macintosh®, UNIX®)
- Provides for robust evolution of component-based applications and systems
- Is extensible

In addition, COM provides mechanisms for the following:

- Communications between components, even across process and network boundaries
- Shared memory management between components
- Error and status reporting
- Dynamic loading of components

It is important to note that COM is a general architecture for component software. While Microsoft is applying COM to address specific areas such as controls, compound documents, automation, data transfer, storage and naming, and others, any developer can take advantage of the structure and foundation that COM provides.

How does COM enable interoperability? What makes it such a useful and unifying model? To address these questions, it will be helpful to first define the basic COM design principles and architectural concepts. In doing so, we will examine the specific problems that COM is meant to solve and how COM provides solutions for these problems.

The Component Software Problem

The most fundamental question COM addresses is: How can a system be designed such that binary executables from different vendors, written in different parts of the world, and at different times are able to interoperate? To solve this problem, we have to find solutions to four specific problems:

- Basic interoperability—How can developers create their own unique components, yet be assured that these components will interoperate with other components built by different developers?
- Versioning—How can one system component be upgraded without requiring all the system components to be upgraded?
- Language independence—How can components written in different languages communicate?
- Transparent cross-process interoperability—How can we give developers the flexibility to write components to run in-process or cross-process (and eventually cross-network), using one simple programming model?

Additionally, high performance is a requirement for a component software architecture. While cross-process and cross-network transparency is a laudable goal, it is critical for the commercial success of a binary component marketplace that components interacting within the same address space be able to utilize each other's services without any undue "system" overhead. Otherwise, the components will not realistically be scalable down to very small, lightweight pieces of software equivalent to C++ classes or graphical user-interface (GUI) controls.

COM Fundamentals

The Component Object Model defines several fundamental concepts that provide the model's structural underpinnings. These include:

- A binary standard for function calling between components.
- A provision for strongly-typed groupings of functions into interfaces.
- A base interface providing:

- A way for components to dynamically discover the interfaces implemented by other components.
- Reference counting to allow components to track their own lifetime and delete themselves when appropriate.

- A mechanism to uniquely identify components and their interfaces.
- A "component loader" to set up component interactions and additionally in the cross-process and cross-network cases to help manage component interactions.

Binary Standard

For any given platform (hardware and operating system combination), COM defines a standard way to lay out virtual function tables (vtables) in memory, and a standard way to call functions through the vtables. Thus, any language that can call functions via pointers (C, C++, Small Talk®, Ada, and even Basic) all can be used to write components that can interoperate with other components written to the same binary standard. The double indirection (the client holds a pointer to a pointer to a vtable) allows for vtable sharing among multiple instances of the same object class. On a system with hundreds of object instances, vtable sharing can reduce memory requirements considerably.

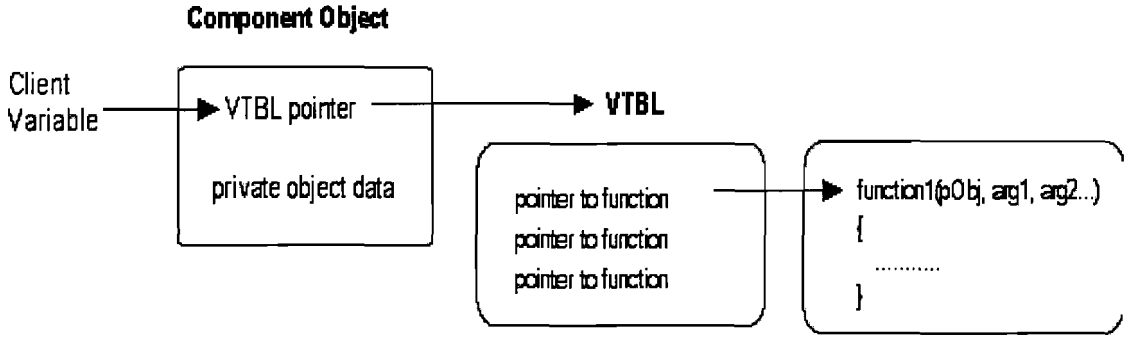

Figure 2. Diagram of a vtable

Objects and components

The word *object* tends to mean something different to everyone. To clarify, in COM, an object is some piece of compiled code that provides some service to the rest of the system. To avoid confusion, it is probably best to refer to a COM object as a "component object" or simply a "component." This avoids confusing COM objects with source-code object-oriented programming (OOP) objects such as those defined in C++. component objects support a base interface called **IUnknown** (described below), along with any combination of other interfaces, depending on what functionality the component object chooses to expose.

Component objects usually have some associated data, but unlike C++ objects, a given component object will never have direct access to another component object in its entirety. Instead, component objects *always* access other component objects through interface pointers. This is a primary architectural feature of the Component Object Model, because it allows COM to completely preserve encapsulation of data and processing, a fundamental requirement of a true component software standard. It also allows for transparent remoting (cross-process or cross-network calling) since all data access is through methods that can exist in a proxy object that forwards the request and vectors back the response.

Interfaces

In COM, applications interact with each other and with the system through collections of functions called *interfaces*. Note that all OLE services are simply COM interfaces. A COM interface is a strongly-typed *contract* between software components to provide a small but useful set of semantically related operations (methods). An interface is the definition of an expected behavior and expected responsibilities. OLE's drag-and-drop support is a good example. All of the functionality that a component must implement to be a drop target is collected into the **IDropTarget** interface; all the drag source functionality is in the **IDragSource** interface.

Interface names begin with "I" by convention. OLE provides a number of useful general purpose interfaces (which generally begin with "**IOle**"), but because anyone can define custom interfaces as well, we expect customers will develop their own interfaces as they deploy component-based applications. Incidentally, a pointer to a component object is really a pointer to one of the interfaces that the component object implements; this means that you can use a component object pointer *only* to call a method, but *not* to modify data, as described above. Here is an example of an interface **ILookup** with two member methods:

```
interface ILookup : public IUnknown
{
   public:
   virtual HRESULT __stdcall LookupByName( LPTSTR lpName,TCHAR **lplpNumber) = 0;

   virtual HRESULT __stdcall LookupByNumber(LPTSTR lpNumber, TCHAR **lplpName) = 0;

};
```

Attributes of interfaces

Given that an interface is a contractual way for a component object to expose its services, there are four very important points to understand:

- **An interface is not a class.** While a class can be instantiated to form a component object, an interface cannot be instantiated by itself because it carries no implementation. A component object must implement that interface and that component object must be instantiated for there to be an interface. Furthermore, different component object classes may implement an interface differently, so long as the behavior conforms to the interface definition (such as two objects that implement **IStack**, where one uses an array and the other a linked list). Thus the basic principle of polymorphism fully applies to component objects.
- **An interface is not a component object.** An interface is just a related group of functions and is the binary standard through which clients and **component object**s communicate. The **component object** can be implemented in any language with any internal state representation, so long as it can provide pointers to interface member functions.
- **Clients only interact with pointers to interfaces.** When a client has access to a component object, it has nothing more than a pointer through which it can access the functions in the interface, called simply an *interface pointer*. The pointer is opaque; it hides all aspects of internal implementation. You cannot see of the component object's data, as opposed to C++ *object pointers* through which a client may directly access the object's data. In COM, the client can call only methods of the interface to which it has a pointer. This encapsulation is what allows COM to provide the efficient binary standard that enables local/remote transparency.
- **Component objects can implement multiple interfaces.** A component object can—and typically does—implement more than one interface. That is, the class has more than one set of services to provide. For example, a class might support the ability to exchange data with clients as well as the ability to save its persistent state information (the data it would need to reload to return to its current state) into a file at the client's request. Each of these abilities is expressed through a different interface (**IDataObject** and **IPersistFile**), so the component object must implement two interfaces.
- **Interfaces are strongly typed.** Every interface has its own interface identifier, a globally unique ID (GUID) described below, thereby eliminating any chance of collision that would occur with human-readable names. The difference between components and interfaces has two important implications. If a developer creates a new interface, she must also create a new identifier for that interface. When a developer uses an interface, he must use the identifier for the interface to request a pointer to the interface. This explicit identification improves robustness by eliminating naming conflicts that would result in run-time failure.
- **Interfaces are immutable.** COM interfaces are never versioned, which means that version conflicts between new and old components are avoided. A new version of an interface, created by adding more functions or changing semantics, is an entirely new interface and is assigned a new unique identifier. Therefore, a new interface does not conflict with an old interface even if all that changed is one operation or semantics (but not even the syntax) of an existing method. Note that as an implementation matter, it is likely that two very similar

interfaces can share a common internal implementation. For example, if a new interface adds only one method to an existing interface, and the component author wishes to support both old-style and new-style clients, she would express both collections of capabilities through two interfaces, but internally implement the old interfaces as a proper subset of the implementation of the new.

It is convenient to adopt a standard pictorial representation for objects and their interfaces. The adopted convention is to draw each interface on an object as a "plug-in jack."

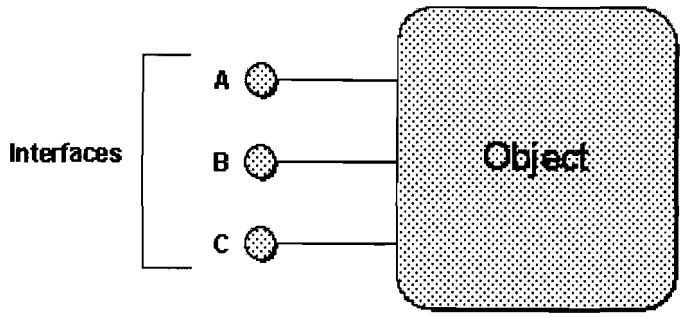

Figure 3. A typical picture of a component object that supports three interfaces A, B, and C.

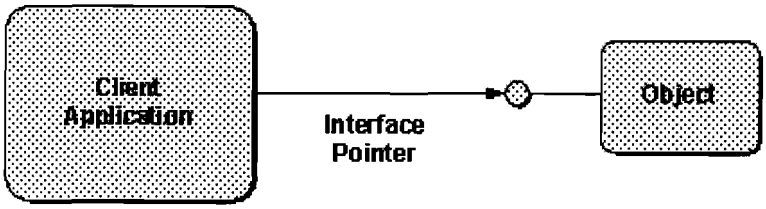

Figure 4. Interfaces extend toward the clients connected to them.

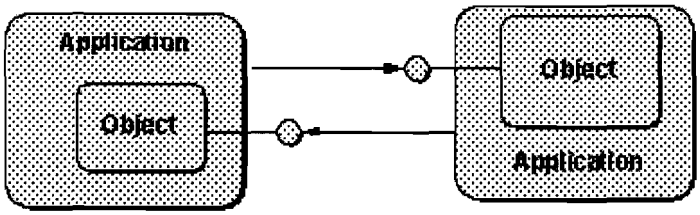

Figure 5. Two applications may connect to each other's objects, in which case they extend their interfaces toward each other.

The unique use of interfaces in COM provides five major benefits:

1. **The ability for functionality in applications (clients or servers of objects) to evolve over time.** This is accomplished through a request called **QueryInterface** that absolutely all COM objects support (or else they are not COM objects). **QueryInterface** allows an object to make more interfaces (that is, support new groups of functions) available to new clients while at the same time retaining complete binary compatibility with existing client code. In other words, revising an object by adding new functionality will not require any recompilation on the part of any existing clients. This is a key solution to the problem of versioning and is a fundamental requirement for achieving a component software market. COM additionally provides for robust versioning because COM interfaces are immutable, and components continue to support old interfaces even while adding new functionality through additional interfaces. This guarantees backward compatibility as components are upgraded. Other proposed system object models, on the other hand, generally allow developers to change existing interfaces, leading ultimately to versioning problems as components are upgraded. While these approaches may appear on the surface to handle versioning, we haven't seen one that actually works—for example, if version checking is done only at object creation time, subsequent users of an instantiated object can easily fail because the object is of the right type but the wrong version (and per-call version checking is too expensive to even contemplate!)
2. **Fast and simple object interaction.** Once a client establishes a connection to an object, calls to that object's services (interface functions) are simply indirect functions calls through two memory pointers. As a result, the performance overhead of interacting with an in-process COM object (an object that is in the same address

space) as the calling code is negligible. Calls between COM components in the same process are only a handful of processor instructions slower than a standard direct function call and no slower than a compile-time bound C++ object invocation. In addition, using multiple interfaces per object is efficient because the cost of negotiating interfaces (via **QueryInterface**) is done in groups of functions instead of one function at a time.

3. **Interface reuse.** Design experience suggests that there are many sets of operations that are useful across a broad range of components. For example, it is commonly useful to provide or use a set of functions for reading or writing streams of bytes. In COM, components can reuse an existing interface (such as **IStream**) in a variety of areas. This not only allows for code reuse, but by reusing interfaces, the programmer learns the interface once and can apply it throughout many different applications.
4. **"Local/Remote Transparency."** The binary standard allows COM to intercept an interface call to an object and make instead a *remote procedure call* to an object that is running in another process or on another machine. A key point is that the caller makes this call exactly as it would for an object in the same process. The binary standard enables COM to perform inter-process and cross-network function calls transparently. While there is, of course, more overhead in making a remote procedure call, no special code is necessary in the client to differentiate an in-process object from out-of-process objects. This means that as long as the client is written from the start to handle remote procedure call (RPC) exceptions, all objects (in-process, cross-process, and remote) are available to clients in a uniform, transparent fashion. Microsoft will be providing a distributed version of COM that will require no modification to existing components in order to gain distributed capabilities. In other words, programmers are completely isolated from networking issues, and indeed, components shipped today will operate in a distributed fashion when this future version of COM is released.
5. **Programming language independence.** Any programming language that can create structures of pointers and explicitly or implicitly call functions through pointers can create and use component objects. Component objects can be implemented in a number of different programming languages and used from clients that are written using completely different programming languages. Again, this is because COM, unlike an object-oriented programming language, represents a binary object standard, not a source code standard.

Globally Unique Identifiers (GUIDs)

COM uses globally unique identifiers—128-bit integers that are guaranteed to be unique in the world across space and time—to identify every interface and every component object class. These globally unique identifiers are UUIDs (universally unique IDs) as defined by the Open Software Foundation's Distributed Computing Environment. Human-readable names are assigned only for convenience and are locally scoped. This helps ensure that COM components do not accidentally connect to "the wrong" component, interface, or method, even in networks with millions of component objects. CLSIDs are GUIDs that refer to component object classes, and IID are GUIDs that refer to interfaces. Microsoft supplies a tool (uuidgen) that automatically generates GUIDs. Additionally, the **CoCreateGuid** function is part of the COM API. Thus, developers create their own GUIDs when they develop component objects and custom interfaces. Through the use of defines, developers don't need to be exposed to the actual 128-bit GUID. For those who want to see real GUIDs in all their glory, the example below shows two GUIDs. **CLSID_PHONEBOOK** is a component object class that gives users lookup access to a phone book. **IID_ILOOKUP** is a custom interface implemented by the PhoneBook class that accesses the phonebook's database:

DEFINE_GUID(CLSID_PHONEBOOK, 0xc4910d70, 0xba7d, 0x11cd, 0x94, 0xe8, 0x08, 0x00, 0x17, 0x01, 0xa8, 0xa3);

DEFINE_GUID(IID_ILOOKUP, 0xc4910d71, 0xba7d, 0x11cd, 0x94, 0xe8, 0x08, 0x00, 0x17, 0x01, 0xa8, 0xa3);

The GUIDs are embedded in the component binary itself and are used by the COM system dynamically at bind time to ensure that no false connections are made between components.

IUnknown

COM defines one special interface, **IUnknown**, to implement some essential functionality. All component objects are required to implement the **IUnknown** interface, and conveniently, all other COM and OLE interfaces derive from **IUnknown**. **IUnknown** has three methods: **QueryInterface**, **AddRef**, and **Release**. In C++ syntax, **IUnknown** looks like this:

```
interface IUnknown {
   virtual   HRESULT   QueryInterface(IID& iid, void** ppvObj) = 0;
   virtual   ULONG   AddRef() = 0;
   virtual   ULONG   Release() = 0;
```

```
}
```

AddRef and **Release** are simple reference counting methods. A component object's **AddRef** method is called when another component object is using the interface; the component object's **Release** method is called when the other component no longer requires use of that interface. While the component object's reference count is nonzero, it must remain in memory; when the reference count becomes zero, the component object can safely unload itself because no other components hold references to it.

QueryInterface is the mechanism that allows clients to dynamically discover (at run time) whether or not an interface is supported by a component object; at the same time, it is the mechanism that a client uses to get an interface pointer from a component object. When an application wants to use some function of a component object, it calls that object's **QueryInterface**, requesting a pointer to the interface that implements the desired function. If the component object supports that interface, it will return the appropriate interface pointer and a success code. If the component object doesn't support the requested interface, then it will return an error value. The application will then examine the return code; if successful, it will use the interface pointer to access the desired method. If the **QueryInterface** failed, the application will take some other action, letting the user know that the desired method is not available.

The example below shows a call to **QueryInterface** on the component PhoneBook. We are asking this component, "Do you support the **ILookup** interface?" If the call returns successfully, then we know that the component object supports the **ILookup** interface, and we've got a pointer to use to call methods contained in the **ILookup** interface (either **LookupByName** or **LookupByNumber**). If not, then we know that the component object PhoneBook does not implement the **ILookup** interface.

```
LPLOOKUP *pLookup;
TCHAR szNumber[64];
HRESULT hRes;

//  Call QueryInterface on the component object PhoneBook, asking for a pointer
//  to the Ilookup interface identified by a unique interface ID.

hRes = pPhoneBook->QueryInterface( IID_ILOOKUP, &pLookup);
if( SUCCEEDED( hRes ) )
{
    pLookup->LookupByName("Daffy Duck", &szNumber); // use Ilookup interface pointer
    pLookup->Release();  // finished using the IPhoneBook interface pointer
}
else
{
   // Failed to acquire Ilookup interface pointer.
}
```

Note that **AddRef** is not explicitly called in this case because the **QueryInterface** implementation increments the reference count before it returns an interface pointer.

Component Object Library

The Component Object Library is a system component that provides the mechanics of COM. The Component Object Library provides the ability to make **IUnknown** calls across processes; it also encapsulates all the "legwork" associated with launching components and establishing connections between components. Typically when an application creates a component object, it passes the CLSID of that component object class to the Component Object Library. The Component Object Library uses that CLSID to look up the associated server code in the registration database. If the server is an executable, COM launches the .EXE file and waits for it to register its class factory through a call to **CoRegisterClassFactory** (a class factory is the mechanism in COM used to instantiate new component objects). If that code happens to be a dynamic-link library (DLL), COM loads the DLL and calls **DllGetClassFactory**. COM uses the object's **IClassFactory** to ask the class factory to create an instance of the component object and returns a pointer to the requested interface back to the calling application. The calling application neither knows nor cares where the server application is run; it simply uses the returned interface pointer to communicate with the newly created component object. The Component Object Library is implemented in COMPOBJ.DLL for Windows and OLE32.DLL for Windows NT and Windows 95.

To summarize, COM defines several basic fundamentals that provide the underpinnings of the object model. The binary standard allows components written in different languages to call each other's functions. Interfaces are logical groups of related functions—functions that together provide some well-defined capability. **IUnknown** is the interface that COM defines to allow components to control their own life span and to dynamically determine another component's capabilities. A component object implements **IUnknown** to control its life span and to provide access to the interfaces it supports. A component object does not provide direct access its data. GUIDs provide a unique identifier for each class and interface, thereby preventing naming conflicts. And finally, the Component Object Library is implemented as part of the operating system and provides the "legwork" associated with finding and launching component objects.

Now that we've got a good understanding of COM's fundamental pieces, let's look at how these pieces fit together to enable component software.

COM Solves the Component Software Problem

COM addresses the four basic problems associated with component software:

- Basic component interoperability
- Versioning
- Language independence
- Transparent cross-process interoperability

In addition, COM provides a high-performance architecture to meet the requirements of a commercial component market.

Basic Interoperability and Performance

Basic interoperability is provided by COM's use of vtables to define a binary interface standard for method calling between components. Calls between COM components in the same process are only a handful of processor instructions slower than a standard direct function call and no slower than a compile-time bound C++ object invocation.

Versioning

A good versioning mechanism allows one system component to be updated without requiring updates to all the other components in the system. Versioning in COM is implemented using interfaces and **IUnknown::QueryInterface**. The COM design completely eliminates the need for things like version repositories or central management of component versions.

When a software module is updated, it is generally to add new functionality or to improve existing functionality. In COM, you add new functionality to your component object by adding support for new interfaces. Since the existing interfaces don't change, other components that rely on those interfaces continue to work. Newer components that know about the new interfaces can use those newly exposed interfaces. Because **QueryInterface** calls are made at run time without any expensive call to some "capabilities database" (as used in some other system object models), the current capabilities of a component object can be efficiently evaluated each time the component is used; when new features become available, applications that know how to use them will begin to do so immediately.

Improving existing functionality is even easier. Because the syntax and semantics of an interface remain constant, you are free to change the implementation of an interface, without breaking other developers components that rely on the interface. For example, say you have a component that supports the (hypothetical) **IStack** interface, which would include methods like Push and Pop. You've currently implemented the interface as an array, but you decide that a linked list would be more appropriate. Since the methods and parameters do not change, you can freely replace the old implementation with a new one, and applications that use your component will get the improved linked list functionality "for free."

Windows and OLE use this technique to provide improved system support. For example, in OLE today, structured storage is implemented as a set of interfaces which currently use the C run-time file input/output functions

internally. In Windows 2000 (the next version of Windows NT), those same interfaces will write directly to the file system. The syntax and semantics of the interfaces remain constant; only the implementation changes. Existing applications will be able to use the new implementation without any changes; they get the improved functionality "for free."

The combination of the use of interfaces (immutable, well-defined "functionality sets" that are extruded by components) and **QueryInterface** (the ability to cheaply determine at run time the capabilities of a specific component object) enable COM to provide an architecture in which components can be dynamically updated, without requiring updates to other reliant components. This is a fundamental strength of COM over other proposed object models. COM solves the versioning/evolution problem where the functionality of objects can change independently of clients of that object without rendering existing clients incompatible. In other words, COM defines a system in which components continue to support the interfaces through which they provided services to older clients, as well as support new and better interfaces through which they can provide services to newer clients. At run time old and new clients can safely coexist with a given component object. Errors can only occur at easily handled times: bind time or during a **QueryInterface** call. There is no chance for a random crash such as those that occur when an expected method on an object simply does not exist or its parameters have changed.

Language Independence

Components can be implemented in a number of different programming languages and used from clients that are written using completely different programming languages. Again, this is because COM, unlike an object-oriented programming language, represents a binary object standard, not a source code standard. This is a fundamental benefit of a component software architecture over object-oriented programming (OOP) languages. Objects defined in an OOP language typically interact only with other objects defined in the same language. This necessarily limits their reuse. At the same time, an OOP language can be used in building COM components, so the two technologies are actually quite complementary. COM can be used to "package" and further encapsulate OOP objects into components for widespread reuse, even within very different programming languages.

Transparent Cross-Process Interoperability

It would be relatively easy to address the problem of providing a component software architecture if software developers could assume that all interactions between components occurred within the same process space. In fact, other proposed system object models do make this basic assumption. The bulk of the work in defining a true component software model involves the transparent bridging of process barriers. In the design of COM, it was understood from the beginning that interoperability had to occur across process spaces since most applications could not be expected to be rewritten as DLLs loaded into shared memory. Also, by solving the problem of *cross-process interoperability*, COM also solves the problem of components communicating transparently between different computers across a network, using the exact same programming interface used for components communicating on the same computer.

The Component Object Library is the key to providing transparent cross-process interoperability. As discussed in the last section, the Component Object Library encapsulates all the "legwork" associated with finding and launching components and managing the communication between components. As shown earlier, the Component Object Library insulates components from the location differences. This means that component objects can interoperate freely with other component objects running in the same process, in a different process, or across the network. The code you use to implement or use a component object in any case is exactly the same. Thus, when a new Component Object Library is released with support for cross-network interaction, existing component objects will be able to work in a distributed fashion without requiring source-code changes, recompilation, or redistribution to customers.

Local/Remote transparency

COM is designed to allow clients to *transparently* communicate with components regardless of where those components are running, be it the same process, the same machine, or a different machine. What this means is that there is a *single programming model* for all types of component objects for not only clients of those component object, but also for the servers of those component objects.

From a client's point of view, all component objects are accessed through interface pointers. A pointer must be in-process, and in fact, any call to an interface function always reaches *some* piece of in-process code first. If the component object is in-process, the call reaches it directly. If the component object is out-of-process, then the call first reaches what is called a "proxy" object provided by COM itself, which generates the appropriate remote procedure call to the other process or the other machine. Note that the client from the start should be programmed to handle RPC exceptions; then it can transparently connect to an object that is in-process, cross-process, or remote.

From a server's point of view, all calls to a component object's interface functions are made through a pointer to that interface. Again, a pointer only has context in a single process, and so the caller must always be some piece of in-process code. If the component object is in-process, the caller is the client itself. Otherwise, the caller is a "stub" object provided by COM that picks up the remote procedure call from the "proxy" in the client process and turns it into an interface call to the server component object.

As far as both clients and servers know, they always communicate directly with some other in-process code, as illustrated in Figure 6.

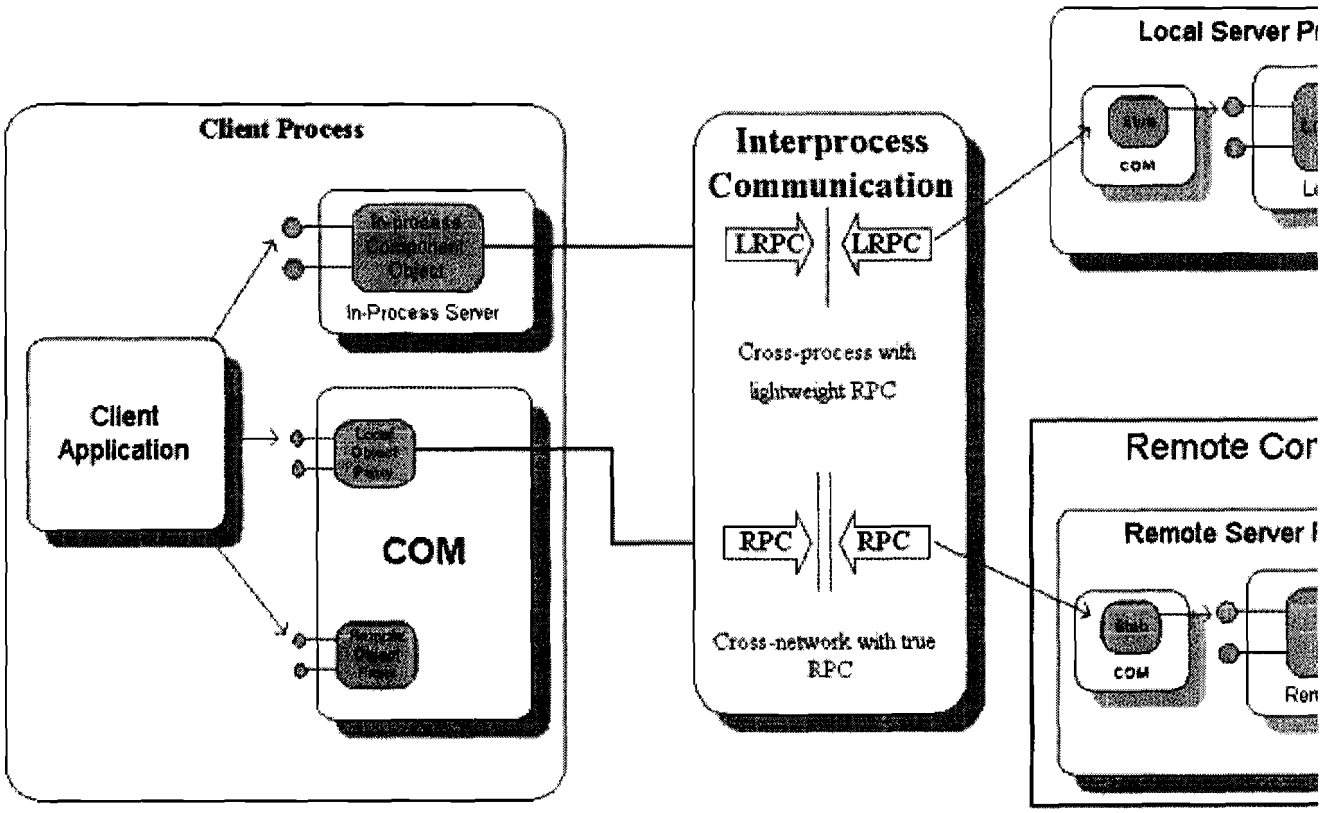

Figure 6. Clients always call in-process code; component objects are always called by in-process code. COM provides the underlying transparent RPC.

The bottom line is that *dealing with local or remote component objects is transparent and identical to dealing with in-process component objects*. This local/remote transparency has a number of key benefits:

- **A common solution to problems that are independent of the distance between client and server.** For example, connection, function invocation, interface negotiation, feature evolution, and so forth occur the same for components interoperating in the same process as for components interoperating across global networks.
- **Programmers leverage their learning.** New services are simply exposed through new interfaces, and once programmers learn how to deal with interfaces, they already know how to deal with new services that will be created in the future. This is a great improvement over environments where each service is exposed in a completely different fashion. For example, Microsoft is working with other independent software vendors (ISVs)

to extend OLE services. These new services, which will be quite diverse in function, will all be very similar in their implementations because they will simply be sets of COM interfaces.

- **Systems implementation is centralized.** The implementers of COM can focus on making the central process of providing this transparency as efficient and powerful as possible, such that every piece of code that uses COM benefits.
- **Interface designers concentrate on design.** In designing a suite of interfaces, the designers can spend their time in the essence of the design—the contracts between the parties—without having to think about the underlying communication mechanisms for any interoperability scenario. COM provides those mechanisms for free, including network transparency.

COM and the Client Server Model

The interaction between component objects and the users of those component objects in COM is in one sense based on a client/server model. We have already used the term "client" to refer to some piece of code that is using the services of an component object. Because a component object supplies services, the implement of that component is usually called the "server"—the component object that serves those capabilities. A client/server architecture in any computing environment leads to greater robustness: If a server process crashes or is otherwise disconnected from a client, the client can handle that problem gracefully and even restart the server if necessary. As robustness is a primary goal in COM, a client/server model naturally fits. Because COM allows clients and servers to exist in different process spaces (as desired by component providers), crash protection can be provided between the different components making up an application. For example, if one component in a component-based application fails, the entire application will not crash. In contrast, object models that are only in-process cannot provide this same fault tolerance. The ability to cleanly separate object clients and object servers in different process spaces is very important for a component software standard that promises to support sophisticated applications.

But unlike other object models we know of, COM is unique in allowing clients to also represent themselves as servers. In fact, many interesting designs have two (or more) components using interface pointers on each other, thus becoming clients and servers simultaneously. In this sense, COM also supports the notion of peer-to-peer computing and is quite different and, we think, more flexible and useful than other proposed object models where clients *never* represent themselves as objects.

Server Flavors: In-Process and Out-of-Process

In general a "server" is some piece of code that implements some component object such that the Component Object Library and its services can run that code and have it create component objects.

Any specific server can be implemented in one of a number of flavors depending on the structure of the code module and its relationship to the client process that will be using it. A server is either "in-process," which means its code executes in the same process space as the client (as a DLL), or "out-of-process," which means it runs in another process on the same machine or in another process on a remote machine (as a .EXE file). These three types of servers are called "in-process," "local," and "remote."

Component object implementers choose the type of server based on the requirements of implementation and deployment. COM is designed to handle all situations from those that require the deployment of many small, lightweight in-process components (like OLE Controls, but conceivably even smaller) up to those that require deployment of a huge components, such as a central corporate database server. And as discussed, all component objects look the same to client applications, whether they are in-process, local, or remote.

Custom Interfaces and Interface Definitions

When a developer defines a new custom interface, he can create an interface definition using the interface definition language (IDL). From this interface definition, the Microsoft IDL compiler generates header files for use by applications using that interface, source code to create proxy, and stub objects that handle remote procedure calls. The IDL used and supplied by Microsoft is based on simple extensions to the Open Software Foundation distributed computing environment (DCE) IDL, a growing industry standard for RPC-based distributed computing.

IDL is simply a tool for the convenience of the interface designer and is not central to COM's interoperability. It

really just saves the developer from manually creating header files for each programming environment and from creating proxy and stub objects by hand. Note that IDL is not necessary unless you are defining a custom interface for an object; proxy and stub objects are already provided with the Component Object Library for all COM and OLE interfaces. Here is the IDL file used to define the custom interface, **ILookup**, which is implemented by the PhoneBook object:

```
[
   object,
   uuid(c4910d71-ba7d-11cd-94e8-08001701a8a3),    // Use the GUID for the Ilookup interface
   pointer_default(unique)
]
interface ILookup : IUnknown          // ILookUp interface derives from IUnkown
{
   import "unknwn.idl";               // Bring in the supplied IUnkown IDL
                                      // Define member function LookupByName:
   HRESULT LookupByName( [in] LPTSTR lpName, [out, string] WCHAR **lplpNumber);
                                      // Define member function LookupByNumber:
   HRESULT LookupByNumber( [in] LPTSTR lpNumber, [out, string] WCHAR ** lplpName);
}
```

COM and Application Structure

COM is not a specification for how applications are structured: It is a specification for how applications interoperate. For this reason, COM is not concerned with the internal structure of an application—that is the job of programmer and also depends on the programming languages and development environments used. Conversely, programming environments have no set standards for working with objects outside of the immediate application. C++, for example, works extremely well with objects inside an application, but has no support for working with objects outside the application. Generally, other programming languages are the same. COM, through language-independent interfaces, picks up where programming languages leave off, providing network-wide interoperability of components to make up an integrated application.

Summary

The core of the Component Object Model is a specification for how components and their clients interact. As a specification, it defines a number of other standards for interoperability of software components:

- A binary standard for function calling between components
- A provision for strongly-typed groupings of functions into interfaces
- A base **IUnknown** interface providing:
  - A way for components to dynamically discover the interfaces supported by other components (**QueryInterface**)
  - Reference counting to encapsulate component lifetime (***AddRef*** and **Release**)
- A mechanism to uniquely identify components and their interfaces (GUIDs)

In addition to being a specification, COM is also an implementation contained in the Component Object Library. The implementation is provided through a library (such as a DLL on Microsoft Windows/Windows NT) that includes:

- A small number of fundamental API functions that facilitate the creation of COM applications, both clients and servers. For clients, COM supplies basic component object creation functions; for servers, it supplies facilities to expose their component objects.
- Implementation locator services through which COM determines from a class identifier which server implements that class and where that server is located. This includes support for a level of indirection, usually a system registry, between the identity of an component object class and the packaging of the implementation such that clients are independent of the packaging (so packaging can change in the future).
- Transparent remote procedure calls when a component object is running in a local or remote server, as illustrated in Figure 6 above.

In general, only one vendor needs to—or should—implement a COM Library for any particular operating system. For example, Microsoft is implementing COM on Windows, Windows NT, and the Apple Macintosh. Other vendors are implementing COM on other operating systems, including specific versions of UNIX. Also, it is important to note that COM draws a very clean distinction between:

- The object model and the wire-level protocols for distributed services, which are the same on all platforms, and
- Platform-specific operating system services (for example, local security and network transports).

Therefore, developers are not constrained into new and specific models for the services of different operating systems, yet they can develop components that interoperate with components on other platforms.

All in all, only with a binary standard on a given platform and a wire-level protocol for cross-machine component interaction can an object model provide the type of structure necessary for full interoperability between all applications and among all different machines in a network. With a binary and network standard, COM opens the doors for a revolution in innovation without a revolution in programming or programming tools.

Appendix 1: The Problem with Implementation Inheritance

Implementation inheritance—the ability of one component to "subclass" or inherit some of its functionality from another component—is a very useful technology for building applications. Implementation inheritance, however, can create many problems in a distributed, evolving object system.

The problem with implementation inheritance is that the "contract" or relationship between components in an implementation hierarchy is not clearly defined; it is implicit and ambiguous. When the parent or child component changes its behavior unexpectedly, the behavior of related components may become undefined. This is not a problem when the implementation hierarchy is under the control of a defined group of programmers who can make updates to all components simultaneously. But it is precisely this ability to control and change a set of related components simultaneously that differentiates an application, even a complex application, from a true distributed object system. So while implementation inheritance can be a very good thing for building applications, it is not appropriate for a system object model that defines an architecture for component software.

In a system built of components provided by a variety of vendors, it is *critical* that a given component provider be able to revise, update, and distribute (or redistribute) his product without breaking existing code in the field that is using the previous revision or revisions of his component. **In order to achieve this, it is necessary that the actual interface on the component used by such clients be crystal clear to both parties.** Otherwise, how can the component provider be sure to maintain that interface and thus not break the existing clients?

From observation, the problem with implementation inheritance is that it is significantly easier for programmers *not* to be clear about the actual interface between a base and derived class than it is to be clear. This usually leads implementers of derived classes to require source code to the base classes; in fact, most application framework development environments that are based on inheritance provide full source code for this exact reason.

The bottom line is that inheritance, while very powerful for managing source code in a project, is not suitable for creating a component-based system where the goal is for components to reuse each other's implementations without knowing any internal structures of the other objects. Inheritance violates the principle of *encapsulation*, the most important aspect of an object-oriented system.

An Example of the Implementation Inheritance Problem

The following C++ example illustrates the technical heart of the robustness problem:

```
class CBase
   {
   public:
      void DoSomething(void) { ... if (condition) this->Foo(); ...}
      virtual   void Foo(void);
   };

class CDerived : public CBase
   {
```

```
public:
   virtual   void Foo(void);
};
```

This is the classic paradigm of reuse in implementation inheritance: A base class periodically makes calls to its own virtual functions, which may be overridden by its derived classes. In practice in such a situation, **CDerived** can become—and therefore often will become—intimately dependent on *exactly* when and under what conditions **Foo** will be invoked by the class **CBase**.

If, presently, all such **Foo** invocations by **CBase** are intended long term as hooks for the derived class, there is no problem. There are two cases, however. Either the implementation of **CBase::Foo** is implemented as:

```
void CBase::Foo(void)
   {
   //Do absolutely nothing
   }
```

or it is not empty.

If the implementation of **CBase::Foo** is *not* empty, it is carrying out some useful and likely needed transformation on the internal state of **CBase**. Thus, it is very questionable whether *all* of the invocations of **Foo** are for the support of derived classes; some of them instead are likely to be only for the purpose of carrying out this transformation. That transformation is part of current implementation of **CBase**.

*Thus, in summary, **CDerived** becomes coupled to details of that current implementation of **CBase**; the interface between the two is not clear and precise.*

Further coupling comes from the fact that the implementation of **CDerived::Foo**, in addition to performing its own role, must be sure to carry out the transformation carried out in **CBase::Foo**. It can do this by itself, reimplementing the code in **CBase::CBase**, but this causes obvious coupling problems. Alternatively, it can itself invoke the **CBase::Foo** method:

```
void CDerived::Foo(void)
   {
   [Do some work]

   ...
   CBase::Foo();
   ...

   [Do other work]
   }
```

However, it is very unclear what is appropriate or possible for **CDerived** to do in the areas marked "Do some work" and "Do other work." What is appropriate depends, again, heavily on the current implementation of **CBase::Foo**.

If, in contrast, **CBase::Foo** *is* empty, we are not likely in immediate danger of surreptitious coupling. In the implementation of **CBase**, invoking **Foo** clearly serves no immediately useful purpose, and so it is likely that, indeed, all invocations of **Foo** in **CBase** are only for the support of **CDerived**. Consider, however, the case that the **CDerived** class is reused:

```
class CAnother : public CDerived
   {
   public:
      virtual   void Foo(void);
   };
```

Though **CBase::Foo** had a trivial implementation, **CDerived::Foo** will not. (Why override it if otherwise?) The relationship of **CAnother** to **CDerived** thus becomes as problematic as the **CDerived-CBase** relationship in the previous case.

This is the architectural heart of the problem observed in practice that leads to a view that implementation inheritance is unacceptable as the mechanism by which independently developed binary components are reused and refined.

COM provides two other mechanisms for code reuse called *containment/delegation* and *aggregation*. Both of these reuse mechanisms allow objects to exploit existing implementation while avoiding the problems of implementation inheritance. See "Appendix 2: COM Reusability Mechanisms" of this article for an overview of these alternate reuse mechanisms.

Appendix 2: COM Reusability Mechanisms

The key point to building reusable components is *black-box reuse,* which means the piece of code attempting to reuse another component knows nothing—and does not need to know anything—about the internal structure or implementation of the component being used. In other words, the code attempting to reuse a component depends upon the *behavior* of the component and not the exact *implementation*. As illustrated in "Appendix 1: The Problem with Implementation Inheritance," implementation inheritance does not achieve black-box reuse.

To achieve black-box reusability, COM supports two mechanisms through which one component object may reuse another. For convenience, the object being reused is called the *inner object* and the object making use of that inner object is the *outer object.*

- **Containment/Delegation.** The outer object behaves like an object client to the inner object. The outer object "contains" the inner object, and when the outer object wishes to use the services of the inner object, the outer object simply delegates implementation to the inner object's interfaces. In other words, the outer object uses the inner object's services to implement some of its own functionality (or possibly all of its own functionality).
- **Aggregation.** The outer object wishes to expose interfaces from the inner object as if they were implemented on the outer object itself. This is useful when the outer object would always delegate every call to one of its interfaces to the same interface of the inner object. Aggregation is a convenience to allow the outer object to avoid extra implementation overhead in such cases.

These two mechanisms are illustrated in Figures 7 and 8. The important part to both these mechanisms is how the outer object appears to its clients. As far as the clients are concerned, both objects implement interfaces *A*, *B*, and *C*. Furthermore, the client treats the outer object as a black box, and thus does not care, nor does it need to care, about the internal structure of the outer object—the client only cares about behavior.

Containment is simple to implement for an outer object. The process is like a C++ object that itself contains a C++ string object. The C++ object would use the contained string object to perform certain string functions, even if the outer object is not considered a string object in its own right.

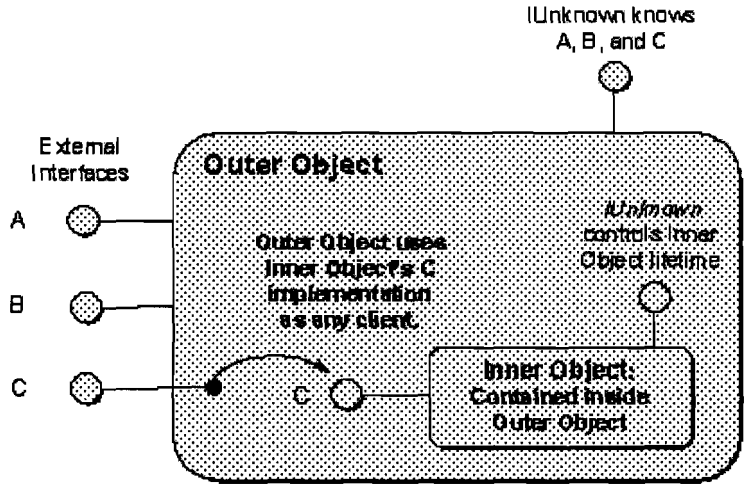

Figure 7. Containment of an inner object and delegation to its interfaces

Aggregation is almost as simple to implement. The trick here is for COM to preserve the function of **QueryInterface** for component object clients even as an object exposes another component object's interfaces as its own. The solution is for the inner object to delegate **IUnknown** calls in its own interfaces, but also allow the outer object to access the inner object's **IUnknown** functions directly. COM provides specific support for this solution.

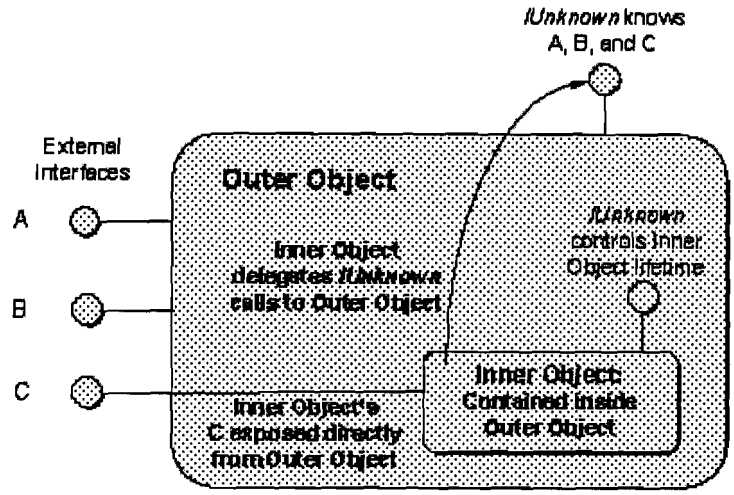

Figure 8. Aggregation of an inner object where the outer object exposes one or more of the inner object's interfaces as its own.

Note The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.